EXCEPCIONES

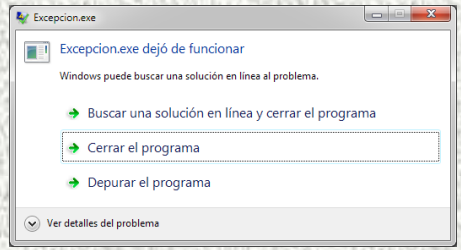
Son errores que se producen durante la ejecución. Si no se implementa un código para que las intercepte, el programa terminará abruptamente (ficheros sin cerrar, pérdida de datos, etc.). Las más habituales son las relacionadas con el manejo de memoria, y no se pueden evitar comprobando el valor del puntero ya que **new** produce el error antes de retornar.

```
int main()
{
   int *x = NULL;
   unsigned int y = 0x7FFFFFFF;  // 2147e6

   x = new int[y];
   if(x)                         // si != NULL
   {
      x[y-100] = 10;
      cout << "Puntero: " << (void *) x << endl;
      delete [] x;
   }
   else
   {
      cout << "Memoria insuficiente." << endl;
   }

    return 0;
}
```

Error en la consola de salida:

```
terminate called after throwing an instance of '..'
   what():  std::bad_alloc

This application has requested the Runtime to
terminate it in an unusual way.
Please contact the application's support team for
more information.
```

El sistema operativo procesa el error mostrando el diálogo:



EXCEPCIONES

C++ proporciona un mecanismo para detectarlas: usando tres palabras reservadas: **try**, **catch** y **throw**. Su uso permite transferir la ejecución del programa desde el punto donde se puede producir el error a un bloque que trate de evitar la terminación anormal del programa.

- **try**: bloque en el que puede surgir una excepción. Una vez lanzada el control se transfiere al bloque catch (no es una llamada a función, al finalizar no se retorna al bloque try).
- **catch**(*excepción*): bloque que maneja la excepción, recibiendo un parámetro que puede ser un código de error o un mensaje para mostrar. Puede existir más de un bloque catch asociado a un try. Si no se produce error y no se ha lanzado alguna excepción este bloque no se ejecuta. El parámetro puede ser de cualquier tipo (int != unsigned int!!!). Para procesar todas las excepciones **sin considerar el tipo** se utiliza **catch(...)**.
- **throw** *excepción*: origina una excepción, provocando el catch. El valor *excepción* se asigna al catch adecuado (identificación del error). Esta sentencia se puede ejecutar desde dentro del bloque try o desde cualquier función invocada en el bloque. Si se lanza una excepción que no está contemplada por un catch el programa terminará en forma anormal.

Si se produce una excepción con el operador new, se realiza un throw de un objeto de la clase **std::bad_alloc**. En general se utiliza sólo el tipo del objeto, sin importar su nombre.

EXCEPCIONES – Ejemplo

```
#include <iostream>

using namespace std;

int main()
{
   int *x = NULL;
   unsigned int y = x7FFFFFFF;   // 2147E6 de enteros

   try
   {
      x = new int[y];
      x[y-100] = 10;
      cout << "Puntero: " << (void *) x << endl;
      delete[] x;
   }
   catch(std::bad_alloc&)
   {  // Los limitadores {} son OBLIGATORIOS
      cout << "Memoria insuficiente\n" << endl;
   }

   return 0;
}
```

Consola de salida (finaliza SIN ERROR):

```
Memoria insuficiente

Process returned 0 (0x0)  execution time : 0.078 s
Press any key to continue
```

Bloque try/catch en una función

Se puede utilizar este mecanismo para controlar el funcionamiento de cualquier operación realizada por una función, no necesariamente relacionada con el manejo de la memoria.

```
#include <iostream>

using namespace std;
void divide(double, double);

int main()
{
   double num, den;

   do
   {
      cout << "Numerador (0 par salir): ";  cin >> num;
      cout << "Denominador: ";  cin >> den;
      divide(num, den);
   } while(num != 0);

   return 0;
}
```

```
void divide(double num, double den)
{
   try
   {
      if( !den )              // controla la división por 0
      {
         throw 0xFF;          // lanza la excepcion con un codigo
      }
      cout<<"\nResultado: "<<num/den<<endl<<endl<<endl;
   }
   catch (int codigo)
   {
      cout<<"\nDivisor 0!!! (codigo = "<<codigo<<").\n\n\n";
   }
}
/*** forma alternativa para utilizar el try/catch ***/
   try
      if( !den )              // controla la division por 0
         throw("\nNo se puede dividir por 0!!!");     // mensaje
    catch(const char *codigo)  // EL MISMO TIPO que throw
      cout << codigo << "\n\n\n";

/***       completar con los limitadores {}       ***/
```

Ej. 02: Division

Múltiples sentencias catch

El tipo de dato enviado con throw se puede utilizar parar usar distintas versiones de la sentencia catch. Las otras opciones de los bloques catch son ignoradas.

```
#include <iostream>
using namespace std;

void Xhandler(int test)
{
   try
   {
      if(test==0)
         throw 5;             // throw un int
      if(test==1)
         throw 'A';           // throw un char
      if(test==2)
         throw 123.45;        // throw un double
   }
   catch(int i)               // catch un int
   {  cout << "Capturado un entero: " << i << "\n\n"; }
   catch(char c)              // catch un char
   {  cout << "Capturado un caracter: " << c << "\n\n"; }
   catch(double d)            // catch un doouble
   {  cout << "Capturado un double: " << d << "\n\n";  }
   catch(...) // PUEDE PROCESAR TODOS LOS TIPOS
   {  cout << "Captura todos los tipos\n\n"; }
}

int main()
{
   cout << "Inicio\n\n";
   Xhandler(0);
   Xhandler(1);
   Xhandler(2);
   cout << "\nFin\n\n";

   return 0;
}
```

Excepciones acotadas

Se puede restringir el tipo de excepciones que una función puede lanzar, incluyendo la cláusula throw en la definición de la misma. No es soportado por todos los compiladores.

<tipo_retorno> <nombre_función>(<lista_parámetros>) **throw** (<lista_tipos>);

Sólo los tipos de datos contenidos en <lista_tipos> (separados por comas) pueden ser utilizados por la función para usar throw y ser alcanzados por el correspondiente catch. Si se lanza una **excepción no prevista** el **programa terminará en forma anormal**. La lista vacía indica que la función no puede producir excepciones. Se verifica en tiempo de ejecución.

```
int Compara(int, int) throw();              // Compara no puede producir excepciones.

int Array(int) throw(std::bad_alloc);   // sólo produce excepciones por memoria insuficiente.

int MiFuncion(char *) throw(std::bad_alloc, ExDivCero);      // puede producir excepciones
                                                    // por falta de memoria o por división por cero.
```

Excepciones acotadas – Ejemplo

```
#include <iostream>

using namespace std;

void Xhandler(int test)  throw(int, char, double)
{
   if(test==0)
      throw 5;              // throw un int
   if(test==1)
      throw 'A';            // throw un char
   if(test==2)
      throw 123.45;         // throw un double
   if(test==3)
      throw "pepe";         // throw un const char*
}
```

```
int main()
{
   cout << "Inicio\n\n";
   try
   {
      Xhandler(3);          // probar los valores, con 3 se cuelga
   }
   catch(int i)
   {
      cout << "Capturado integer\n\n";
   }
   catch(char c)
   {
      cout << "Capturado char\n\n";
   }
   catch(double d)
   {
      cout << "Capturado double\n\n";
   }
   catch(...)
   {
      cout << "Capturado otro tipo\n\n";
   }
   cout << "\nFin\n\n";

   return 0;
}
```

Ej. 04: Exc acotadas

Clase exception

En el fichero de cabecera <**exception**>, incluído en el estándar std, se declara la clase base *exception* a partir de la cual se pueden derivar clases que pueden ser utilizadas como un nuevo tipo para lanzar excepciones específicas que encapsulan la información correspondiente al error.

```
class exception
{
 public:
      exception() throw() { }
       virtual ~exception() throw();
       virtual const char* what() const throw();
};
```

La cláusula *throw* utilizada en los métodos indica que los mismos no pueden producir ningún tipo de excepción. La función *what* retorna una cadena para indicar el motivo de la excepción.

Clase exception – Ejemplo

```
#include <iostream>
#include <fstream>
using namespace std;

class ExcepDeriv : public exception    // derivada
{
  public:
     ExcepDeriv(int mot) : exception(),
                           motivo(mot) {}
     const char* what() const throw();
  private:
     int motivo;      // código del error
};

const char* ExcepDeriv :: what() const throw()
{
  switch(motivo)
  {
     case 1:
        return "Fichero de origen no existe";
     case 2:
        return "No es posible abrir salida";
  }
  return "Error inesperado";
}
```

```
void CopiaFichero(const char* Origen, const char *Destino)
{
   unsigned char buffer[1024];
   int leido;
   ifstream fe(Origen, ios::in | ios::binary);
   if( !fe.good() )
      throw ExcepDeriv(1);    // se produce error en la entrada
   ofstream fs(Destino, ios::out | ios::binary);
   if( !fs.good() )
      throw ExcepDeriv(2);    // se produce error en la salida
   do
   {
      fe.read(reinterpret_cast<char *>(buffer), 1024);
      leido = fe.gcount();
      fs.write(reinterpret_cast<char *>(buffer), leido);
   } while( leido );
   fe.close();    fs.close();
}

int main()
{
   char Desde[] = "excepcion.cpp";
   char Hacia[] = "excepcion.cpy";
   try { CopiaFichero(Desde, Hacia);  }
   catch(ExcepDeriv &ex) { cout << ex.what() << endl;  }
   return 0;
}
```

Ej. 05: Exc file

Excepciones estándar

Existen cuatro excepciones estándar, derivadas de la clase *exception*, y asociadas a un operador o a un error de especificación:

std::bad_alloc	// operador new
std::bad_cast	// operador dynamic_cast<>
std::bad_typeid	// operador typeid
std::bad_exception	// cuando se viola una especificación

Cada vez que se utiliza uno de estos operadores, puede producirse una excepción. Un programa bien realizado debería prever el tratamiento de estas excepciones para crear aplicaciones robustas y seguras.

Excepciones en constructores y destructores

Generalmente en los constructores es donde se requiere de un mayor tratamiento de excepciones. En ellos se realizan las peticiones de memoria; la inicialización de las variables a partir de un fichero de configuración, etc.

En cambio, si bien está permitido, no se aconseja producir excepciones en los destructores. Si durante el procesamiento de una excepción se invoca automáticamente un destructor y el mismo produce una nueva excepción, el programa terminará inmediatamente.

Existe una función estándar, declarada en <**exception**>, **uncaught_exception** que retorna true si se está procesando una excepción, que puede utilizarse para prevenir la ejecución de la sentencia throw.

Relanzar una excepción

El bloque catch puede relanzar una excepción con la sentencia throw sin argumentos.

```
#include <iostream>
#include <fstream>
using namespace std;

enum acceso {IN, OUT};
void Xhandler(const char* , acceso, fstream*);
void Backup(void);

int main()
{
   // inicio del programa
   try  {
      Backup();
   }
   catch (acceso valor)
   {
      cout<<"\n\nManejo de la exepcion relanzada:\n";
      if(valor)
         cout << "\tError en el fichero de salida\n";
      else
         cout << "\tError en el fichero de origen\n";
      // rutina de manejo del error relanzado
   }
   // resto del programa sin error
   return 0;
}
```

```
void Xhandler(const char* nom, acceso modo, fstream *fh)
{
   if (modo)                    // escritura
      fh->open(nom, ios_base::out | ios_base::binary);
   else                         // lectura
      fh->open(nom, ios_base::in | ios_base::binary);
   if( !fh->good() )
      throw modo;          // en caso de error se lanza la exepción
}
 void Backup(void)
{
   char Origen[] = "datos.txt“, Destino[] = "excepcion.cpy";
   fstream *fin = new fstream, *fout = new fstream;
   try {
      Xhandler(Origen, IN, fin);
      Xhandler(Destino, OUT, fout); }
   catch (acceso valor) {
      if(valor) {
         cout << "El fichero de salida no se puede abrir\n\n";
         fin->close(); }
      else
         cout << "El fichero de origen no existe\n\n";
      throw ;  // relanzo para comunicar al main() }
   /* resto del programa */ ;  fin->close();     fout->close();
}
```

Funciones terminate() y unexpected()

Parte de la Librería Estándar de C++, sus prototipos se incluye la cabecera <**exception**>.

terminate es invocada cuando a) no se encuentra un catch que corresponda con el tipo de excepción lanzada, b) el programa trata de relanzar una excepción que no ha sido lanzada originalmente, c) se presentan otras circunstancias especiales (mal manejo del stack, que un destructor que se está ejecutando lance una excepción, etc.). Por defecto terminate() invoca a **abort**(), pero se puede variar llamando a la función **set_terminate**().

unexpected es invocada cuando una función intenta lanzar una excepción que no figura como alguno de los parámetros de throw. Mediante la función **set_unexpected**() se puede especificar la función que debe llamar unexpected(), que por defecto invoca a terminate().

Las funciones set_terminate y set_unexpected tienen el prototipo:

[*terminate*|*unexpected*]_*handler* set_[terminate|unexpected]([*ter*|*unex*]_*handler* n) throw ();

Ambas funciones retornan el handler anterior y reciben como parámetro al nuevo handler. Los handlers son **punteros a funciones**, de la forma:

typedef void (*[terminate|unexpected]_handler)();